Intro to Computational Complexity Theory

Suhas Arehalli

COMP221 - Spring 2024

1 Intro to Computational Complexity

- So far, we've been worried about how fast particular algorithms are.
  - But often, we want to know how fast *any* solution to that problem can be.
  - i.e., "Could I write a faster algorithm to solve this problem?"
- In general, it's hard to prove that no faster solution is possible!
  - These are often trivial or near-impossible!
  - i.e., search for unsorted arrays can't be faster than linear time, since we must at least check each index.
  - A more sophisticated argument of this kind is the problem from HW2 where you proved *all* comparison-based sorts are $\Omega(n \log n)$!
- Instead, we'll adopt a *relative* strategy; we will prove a particular problem is at least as hard as some other canonical problem that's known to be difficult!

1.1 (Turing/Cook) Reductions

- Consider the following scenario:
  - We have some problem specification X that we want to prove is hard (i.e., doesn't have a fast solution).
  - Assume we also have some problem specification Y that we *know* is hard — i.e., we know *every* solution to Problem Y is in $\Omega(h(n))$.
  - Let ProbX be an arbitrary algorithm that solves problem X that runs in worst-case time complexity $\Theta(f(n))$
  - Now consider a function Translate that turns a particular *solution* of problem X into a solution for problem Y in time $\Theta(g(n))$.
  - We can then construct a solution to Problem Y in terms of ProbX:

```
function ProbY(...)
    x ← ProbX(...)
    return Translate(x)
```

end function

- Consider the time complexity of this algorithm: $\Theta(f(n) + g(n))$!
- Since this is a solution to Problem Y, and we know Problem Y is hard, we know this solution is $\Omega(h(n))$.
- Thus we know that $f(n) + g(n) \in h(n)$!
- In the best case, $g(n)$ is dominated by $f(n)/h(n)$, and we can conclude that $f(n) \in \Omega(h(n))$!

- Let's sit with that for a second: If we find a sufficiently fast translation from a solution to Problem X to a solution to Problem Y, **and** we know Problem Y is sufficiently hard, then we know that Problem X is also hard!
  - Hard, so far, means that any algorithm that solves the problem is slower than some bound. We'll hone in on what that bound is in a bit!

- Note that I've elided translation of the *inputs* of the problem — this is actually perhaps the *most* interesting part of the translation process, but bear with me for now!

- For now, we're interested in writing functions like ProbY that use calls to algorithms that solve Problem X (i.e., ProbX) in order to solve Problem Y.
  - This is called a *reduction* of Problem Y to Problem X.
  - Note the ordering here! It's tricky!
    * *Reducing* Problem Y to Problem X means *translating* a solution (or multiple solutions) to an instance of Problem X into a solution to Problem Y.
    * A fast reduction of Problem Y to Problem X proves that Problem X is at least as hard as Problem Y.
  - More technically, the general class of reductions we're considering right now are called *Turing* or *Cook Reductions.* We'll look at a different, more specific kind of reduction later.

1.2 Decision Problems

- In general, it's messy to trying to work with the various kinds of problem specifications we've seen so far.

- In the future, we will discuss complexity theory in terms of *Decision Problems* — problems with yes/no (i.e., true or false) answers.

- We do this because of these kinds of problems are well studied and have a lot of theory around them (i.e., from automata/formal language theory!).
  - To fully understand what's going on here, you should take Theory of Computation!
  - For now, you'll see this pay dividends when we talk about Karp/Many-One reductions, and in general when these problems are just easier to work with.

- The claim underlying this choice is that we can convert interesting algorithmic problem specifications into equally interesting (i.e., just as hard!) decision problem variants!
- To see this, consider the case of *Vertex Cover*:
  - A *Vertex Cover* for a Graph $G = (V, E)$ is a set $C \subseteq V$ such that $\forall e \in E$, some vertex $v \in C$ is incident to e. That is, $\forall (v, w) \in E$, $v \in C$ or $w \in C$ (or both!).
  - **(Minimal) Vertex Cover**: Given a Graph $G = (V, E)$, find the *minimal* vertex cover $C \subseteq V$.
  - **Vertex Cover (Decision)**: Given a Graph $G = (V, E)$ and $k \in \mathbb{Z}_{\geq 0}$, determine whether there is a vertex cover C with $|C| \leq k$
- My claim: We can reduce Minimal Vertex Cover to it's decision variant and vice-versa.
- Consider the following reduction:

```
function VertexCover_Decision(G = (V, E), k)
    C ← VertexCover(G)
    if |C| ≤ k then
        return true
    end if
    return false
end function
```

- This reduces the decision problem to the optimization problem with a constant time transformation of the output!
  - This means that if VertexCover runs in $\Theta(f(|V|))$, then VertexCover_Decision can be solved in $\Theta(f(|V|))$ as well!
  - That is, we showed that VertexCover is at least as hard as VertexCover_Decision!
- Of course, this is not surprising: Most of you probably expected that the decision variant was easier than the optimization version!
- So let's consider the trickier reduction:[1]

```
function VertexCover(G = (V, E))
    k ← 0
    while ¬VertexCover_Decision(G, k) do
        k ← k + 1
    end while
    C ← ∅
    for v ∈ V do
        E′ ← E \ {e ∈ E | v incident to e}
        V′ ← V \ {v}
        G′ ← (V′, E′)
        if VertexCover_Decision(G′, k − 1) then
```

[1] This is different than the algorithm I gave you in class — this one is more straightforward, and doesn't require some of the handwaving I did about modifying the edge set!

$C \leftarrow C \cup \{v\}$
$E \leftarrow E'$
$V \leftarrow V'$
$k \leftarrow k - 1$
if $k == 0$ **then**
  return C
end if
end if
end for
end function

- To see that this algorithm is correct, observe...
  - ...that the first for-loop will set k to the size of the minimum vertex cover
  - ...that if we enter the if-statement, the minimum vertex cover of size $k-1$ for G' (call it C') covers all edges that v is no incident to!
  - Thus, $C' \cup \{v\}$ is a vertex cover of G of size k!
  - i.e., at each iteration, $C \subseteq C^*$, where C^* is a minimal vertex cover of (initial) G.
  - Write out a formal proof as practice!
- Assume that VertexCover_Decision runs in $\Theta(f(|V|))$ time. Then this algorithm runs in $\Theta(|V|f(|V|))$ time!
- Is this good enough?
  - This is a *weird* reduction — we call the decision problem's algorithm order $|V|$ times!
  - In terms of asymptotic time complexity, this reduction is an order of magnitude slower!
  - I'll still claim that this difference doesn't matter *for the kinds of things we care about*!
- What do we care about? *Tractability*!
- A problem X is *tractable* if there exists an algorithm that solves that problem in polynomial time. That is, for some $c \in \mathbb{Z}$, there exists a $O(n^c)$ algorithm that solves problem X!
  - Note that this means that an algorithm is *intractable* if there exist **no** polynomial time solutions.
  - Proving tractability means finding a polynomial time algorithm to solve X. Proving intractability means proving **all** algorithms that solve problem X run slower than polynomial time!
- Note that, given this second reduction, we can still claim that if VertexCover_Decision is tractable, then VertexCover is tractable!
  - Perhaps more importantly, this reduction tells us that if VertexCover is *in*tractable, then VertexCover_Decision is also intractable!
- With both reductions, we know that, by our tractability-based definition of hardness, both problems are equally hard!
- Thus, moving forward, we'll be working nearly exclusively with decision problems rather than more complex problem specifications.

2 Reductions & Canonical NP Problems